The Mysteries of Lisp – I: The Way to S-Expression Lisp

Hong-Yi Dai

May 28, 2015

Abstract

Despite its old age, Lisp remains mysterious to many of its admirers. The mysteries on one hand fascinate the language, on the other hand also obscure it. Following Stoyan but paying attention to what he has neglected or omitted, in this first essay of a series intended to unravel these mysteries, we trace the development of Lisp back to its origin, revealing how the language has evolved into its nowadays look and feel. The insights thus gained will not only enhance existent understanding of the language but also inspires further improvement of it.

1 Introduction

> You have to know the past to understand the present.
>
> Carl Sagan

Graham [2001] uncovered "the roots of Lisp",[1] and in particular, showed us "the surprise"[2] that a meta-circular interpreter for the language can be readily constructed. This surprising result, which he nominated as "the defining quality of Lisp", that the language "can be written in itself" [ibid, p. 1], has attracted a lot of language enthusiasts to Lisp. However, when they ask about the source of this surprise, they get answers such as that the language is Turing-complete or that programs are (manipulable as) data in the language. These answers, although succinct, are nebulous. To a large extent, the language remains mysterious.

We intend in a series of essays to unravel the mysteries of Lisp. In the past decades, some scholars have tried to do so and achieved the goal to various degree. The most remarkable is probably Stoyan who carefully studied the history of Lisp for better understanding the language [1979, 1984, 1991, 2007,

[1] Throughout this text, we will use the modern name Lisp for the language. However, when mentioning historical dialects or quoting text about them, we will stick to the ancient name LISP.

[2] The one who first did so was of course McCarthy [1959].

2008]. Holding the same position, we believe that to unravel the mysteries of Lisp, we must trace its development back to its origin. So this work could be considered a continuation of that by Stoyan. We believe most of what we discuss here and will discuss in the following series is folklore knowledge. Moreover, it must have already been investigated by Stoyan. However, we intend not to repeat what Stoyan has done, but to complement his work by gathering what he has neglected or omitted. Our main contribution is exhibiting our findings in one place and interpreting them in both historical and modern contexts.

In this first essay of the series, we will look into the early development of Lisp. We will in particular lay out how the language has evolved into its nowadays look and feel. On just these aspects, a few pointers suffice to show that our grasp of the language is not thorough: we may have heard that Lisp became based on S-expressions more by accident, but have not considered any less-accidental factor; we may have learned that lists are constructed from pairs in Lisp, but not questioned the rationality of this actuality; we may have read that Lisp used to have a kind of expressions other than S-expressions for writing programs, but not investigated the context of their existence.

As far as we know, these issues have not, if ever, been satisfactorily addressed. A probable reason is that most of us never go beyond the landmark paper [McCarthy 1960] that systematically described the language. This paper, however, was not the first systematic description of Lisp. Its earlier draft [McCarthy 1959], published as an AI memorandum, was. It is this AI memo on which we will focus our attention, and prior AI memos in which we will seek useful clues. In our opinion, McCarthy presented in [1959] a better-designed system than the later-determined version in [1960], which evidences again that the development of a system may not necessarily be an advancement but rather a regression.

2 Toward S-expression Lisp

In this section, we will look into the early development of Lisp, to a large extent in chronological order. We will focus our attention mainly on material related to the three issues we set out to address. For a more-general treatment of the early history of Lisp, the reader is referred to either Stoyan [1979; 1984; 1991; 2007; 2008] or McCarthy [1978; 1980].

2.1 An algebraic language

Although according to Stoyan [1984], the incubation of Lisp could date back to 1956, it was in 1958 when the 1st AI memo was published that "an algebraic language for the manipulation of symbolic expressions" (not named LISP yet) [McCarthy 1958a] hatched out. McCarthy put in his new algebraic language most of the features he proposed "for the Volume 2 (V2)" of the International Algebraic Language[3] [1958c]. Among these features, of particular interest to us

[3]IAL, later known as ALGOL, the ALGOrithmic Language

is the proposal of an intermediate language that uniforms function position: a function-designating expression, even an operator, appears always in the head position, as in $f(e_1, \ldots, e_n)$. Another intriguing feature is representing both sequences and expressions as lists implemented using series of machine words on IBM 704. From this vivid description, we can already see a prototype of Lisp. However, note a few things.

First, list was introduced as a kind of *data structure*, not yet abstracted as a *data type*. We diverge from Stoyan on this point. Stoyan held that "lists were not regarded as data structures" [1984, p. 304]. We believe they were, since "a number of interesting and useful operations on lists have been defined" [McCarthy 1958a, p. 5]. They were used for constructing lists and selecting components. The omission of lists "as a kind of quantity"[4] was because "most of the calculations we actually perform cannot as yet be described in terms of these operations" and "it still seems to be necessary to compute with the addresses of the elements of the lists" [ibid, p. 5]. Although it is not clear what McCarthy meant by "cannot as yet be described" here, it is obvious that he felt that these operations were too low-level.

Second, symbolic expressions were not part of the algebraic language. They belonged in both the intermediate language and the language of discourse (English plus mathematics). McCarthy here changed for symbolic expressions from the function notation $f(e_1, \ldots, e_n)$ in [1958c] to the sequence notation $(f, e_1, \ldots, e_n)$, which resembled the prefix notation except explicit parenthesization.

Third, the algebraic language itself used mixfix notation. Although not explicitly stated in [McCarthy 1958a], according to the proposal in [McCarthy 1958c], programs in the algebraic language were supposed to be translated into the intermediate language and then further translated into the assembly language or machine language. Thus the first-stage translation will turn algebraic expressions in mixfix notation completely into symbolic expressions in prefix notation.

The algebraic language received two revisions documented respectively in AIM-3 [McCarthy 1958d] and AIM-4 [McCarthy 1958b]. AIM-3 made explicit that IBM-704 word sequences were internal representations of algebraic expressions while symbolic expressions external. Following the distinction was an attempt to formally define symbolic expressions, spontaneously called "external expressions":[5]

1. A symbol is an [external] expression.
2. If $e_1, e_2, \ldots, e_n$ are [external] expressions, so is $(e_1, e_2, \ldots, e_n)$.

McCarthy further noted the distinction between e and (e). But the special case for empty sequence was missing. Still, symbolic expressions were not admitted into the algebraic language. However, operations on lists were distilled into

[4] In the context, the word 'quantity' probably meant datum or literal. Taking into account the absence of symbolic expressions from the algebraic language (discussed soon), this seems a plausible interpretation.

[5] Our edits are put in square brackets.

more or less `cons`, `car` and `cdr`. In AIM-4, the name LISP, for "List Processor" [McCarthy 1958b, p. 9], first occurred.

2.2 A list processor

In the 8th AI memo [McCarthy 1959], the draft of the landmark paper [McCarthy 1960], symbolic expressions finally entered the algebraic language. Lisp began to feature two systems of notation at the source level: *S-expressions* (short for Symbolic expressions) and *F-expressions*[6] (for Functional expressions), which respectively correspond to the source-level forms of *data* and *programs*.

In this memo, McCarthy presented a system different from what was later given in [1960], and thus even different from what we know today. The definition of S-expressions [McCarthy 1959, p. 3], now complete, is quoted below:[7]

1. The atomic symbols [...] are S-expressions.
2. A null expression [`()`] is also admitted.
3. If e is an S-expression[,] so is `(`e`)`.
4. If e_1 and `(`[e_s]`)` are S-expressions[,] so is `(`e_1`,`[e_s]`)`.

e_s in the 4th clause might refer to a sequence of S-expressions "e_2`,` ... `,`e_n" where $n \geq 2$. In that case, the rule gives us `(`e_1`,`e_2`,` ... `,`e_n`)`. This notation for lists is almost the same as what we know today except that it used commas rather than merely spaces to separate list elements.

According to this definition, valid compound S-expressions include only what we now call *proper lists* that always terminate with `()`, *no* ordered pairs, and naturally *nor* improper lists that do not end with `()`.[8]

As regards the notation for functional expressions, McCarthy switched to square brackets and semi-colons "since parentheses and commas have been preempted" by S-expressions [1959, p. 3]. This switch was a move away from the familiar mathematical notation used since AIM-1, where `f(`e_1`,` ... `,`e_n`)` were used for functional expressions, and `(`e_1`,` ... `,`e_n`)` for symbolic expressions.[9] Had McCarthy retained the notation in AIM-1 or reversed the notation in AIM-8 for S-expressions and F-expressions[10] — in other words, if he used square brackets and semi-colons for S-expressions while parentheses and commas for F-expressions,

[6] 'F-expression' [McCarthy 1959, p. 13] was the original name of M-expressions (for Meta-expressions) [McCarthy 1960, p. 187]. We choose the old name for two reasons: (1) *meta* is relative, the name M-expression becomes misleading once F-expressions are taken to the level of object language; (2) the term F-expression clearly indicates that functional expressions describe functions of S-expressions, which McCarthy called S-functions [1959, p. 1]. Nevertheless, in directly quoted text, we will keep the terms used in the source.

[7] We have slightly edited the original definition to fit modern typographic style. In particular, we use `()` in place of the hand-written Λ for the null expression. Again, our edits are put in square brackets.

[8] From now on, when the word 'list' occurs without any qualifier, it means proper list, as has already been the case.

[9] We could not see how this notation might cause any serious problem.

[10] Interestingly, a later Lisp dialect called M-LISP, which his inventor advertised as a "hybrid of McCarthy's original M-expression LISP and Scheme", did reverse the notation for S-expressions and F-expressions [Muller 1991; 1992].

that is, `[`e_1`;`$\ldots$`;`e_n`]` instead of `(`e_1`,`$\ldots$`,`e_n`)`, and `f(`e_1`,`$\ldots$`,`e_n`)` rather than `f[`e_1`;`$\ldots$`;`e_n`]` — Lisp would have a more-mathematical flavor, which would in turn better justify its being roughly a language for "a mathematical theory of computation" [1961] based on "recursive functions of symbolic expressions" [1959] and surely "an algebraic language for the manipulation of symbolic expressions" [1958a].

As McCarthy noted in the abstract, AIM-8 contained "only the machine[-]independent parts of the system". We see for the first time lists be presented without mentioning memory addresses. In other words, in AIM-8, list got abstracted as a data type. Operations on lists were defined by axioms. The definitions of the selector functions (called `first` and `rest`[11] rather than `car` and `cdr`) and the constructor function (called `combine` instead of `cons`) [McCarthy 1959, pp. 3–4], are reproduced below:

$$\begin{aligned}
\texttt{first[(}e\texttt{)]} &= e \\
\texttt{first[(}e_1\texttt{,}e_s\texttt{)]} &= e_1 \\
\texttt{rest[(}e\texttt{)]} &= \texttt{()} \\
\texttt{rest[(}e_1\texttt{,}e_s\texttt{)]} &= \texttt{(}e_s\texttt{)} \\
\texttt{combine[}e\texttt{;()]} &= \texttt{(}e\texttt{)} \\
\texttt{combine[}e_1\texttt{;(}e_s\texttt{)]} &= \texttt{(}e_1\texttt{,}e_s\texttt{)}
\end{aligned}$$

where e_s might be "$e_2\texttt{,}\ldots\texttt{,}e_n$" for $n \geq 2$. McCarthy further noted that `first` and `rest` are defined only for S-expressions "which are neither null nor atomic", and that `combine` is defined when e_s is not atomic. Note that the S-expression `()`, which represents a null list, was *not* considered an atomic symbol. The constraint on the second argument of `combine` prevents the construction of pairs, and in turn improper lists.

Renaming the selector functions shows that McCarthy "felt uneasy with the machine[-]dependent names" [Stoyan 1991, p. 416] already in use since [1958a]. Constraining the the second argument of the constructor function to lists suggests that he probably recognized the possible misuse of the too-liberal constructor function to build improper lists.

The attempt to rename the selector functions failed and McCarthy reverted to the cryptic names `car` and `cdr`, as "the LISP community was already more powerful [than] the designer" [Stoyan 1991, p. 416]. The attempt to constrain the constructor function, was also abandoned, in our opinion, for no good reason as well.

2.3 A symmetric variant

At the end of AIM-8, McCarthy proposed "binary Lisp" which was a variant that admits "only two[-]element lists" [1959, p. 17]. In particular, the status of

[11] These two names were reintroduced by the PLT people (`http://racket-lang.org/people.html`) into their variant of Scheme (now called Racket, `http://racket-lang.org/`) for `car` and `cdr` constrained (by their contract system) to accepting only proper lists as valid input.

`first` and `rest` was symmetric by definition:

$$\begin{aligned}\texttt{first[(}e_1\texttt{,}e_2\texttt{)]} &= e_1\\ \texttt{rest[(}e_1\texttt{,}e_2\texttt{)]} &= e_2\\ \texttt{combine[}e_1\texttt{;}e_2\texttt{]} &= \texttt{(}e_1\texttt{,}e_2\texttt{)}\end{aligned}$$

What this definition would give is surely *not* "two-element lists" according to the definition given earlier, and had better be called pairs. McCarthy obviously abused the notation for lists here.

Right below the presentation of this definition, McCarthy remarked, in this binary variant, that only two predicates `=` (symbol equality) and `atom` are needed (`null` dropped), and that "the null list can be dispensed with" [ibid, p. 17]. This should not be interpreted as that he tried to ditch the notion of null list because a mathematician knows well the importance of *null* and in [1960] he introduced `NIL` exactly for it. The correct interpretation is that he proposed eliminating `()` as a separate case in the definition of S-expressions, and treating it simply as an atomic symbol. McCarthy further pointed out that "the system is easier until we try to represent functions by expressions [...]" [ibid, p.17]. Here, he probably meant that the system would lose its easy feel to the verbose nesting of pairs for building lists to represent F-expressions.

The system given in [McCarthy 1960] turned out to be exactly this binary variant, albeit reverted to the function names `car`, `cdr` and `cons`. There McCarthy presented the simplified definition of S-expressions [ibid, p. 187] as we know today:

1. Atomic symbols are S-expressions.
2. If e_1 and e_2 are S-expressions, so is $\texttt{(}e_1\texttt{.}e_2\texttt{)}$.

He also resolved all the issues regarding binary Lisp as we see now and he saw then. The two components of a pair was separated by a dot instead of a comma. The atomic symbol `NIL` was chosen to mark the end of a list. The list notation $\texttt{(}e_1\texttt{,}e_2\texttt{,}\ldots\texttt{,}e_n\texttt{)}$ was defined as syntactic sugar for $\texttt{(}e_1\texttt{.(}e_2\texttt{.(}\ldots\texttt{(}e_n\texttt{.NIL)}\ldots\texttt{)))}$.

The system indeed feels simpler. However, it also exposes the underlying representation of lists. Naturally, the constraint on the constructor function of lists was abandoned so as to allow the construction of pairs, and the unconstrained `cons` which mirrors the blind behavior of the corresponding machine instruction returned. As a consequence, improper lists found their way back. Every function expecting a list as argument now should test the argument to see if it is indeed proper, otherwise, it would fail when it receives instead an improper list or a circular list. But given that improper lists are seldom used and the test usually has a linear-time complexity, Lisp programmers either leave it out and assume the input to be a proper list by wishful thinking, or treat the last `cdr` as `NIL` in the case of an improper list and let the trap into an endless loop open in the case of a circular list.

2.4 F-expressions vs. S-expressions

McCarthy himself always preferred and expected to "[write] programs as M-expressions"[12] [1978, p. 179]. Before the first implementation of Lisp came out, programs were indeed written as F-expressions and then hand-compiled to assembly code.[13] These two facts give us a good reason to believe that the implementation McCarthy expected was a compiler that compiles Lisp programs directly or indirectly via S-expressions to assembly code. Our belief is also supported by McCarthy's own words. His remark "you are confusing theory with practice" [Stoyan 1984, p. 307] on Russell's proposal of programming the universal function[14] in an assembly language by hand[15] suggests that, at that time he was not immediately aware that Russell had proposed an implementation of Lisp by interpreting intermediate program representations in the form of S-expressions. Under Russell's proposal, once source programs (in the form of F-expressions) are translated into intermediate programs (completely S-expressions) following the rules of translation first described informally in [McCarthy 1959] and later specified formally in [McCarthy 1960], the hand-compiled implementation of the universal function could readily interpret them. McCarthy later did realize that what Russell obtained by hand-compilation of the universal function "certainly was" a Lisp interpreter [Stoyan 1984, p. 307].

What followed, which was probably one of the most dramatic events in the history of programming languages, that early adopters of Lisp went ahead programming in the intermediate language of S-expressions rather than the source language of F-expressions, was totally against McCarthy's expectation! There might be technical reasons (for example, the translation scheme from F-expressions to S-expressions was not implemented yet) or historical factors (for instance, Russell advertised his result as an interpreter for Lisp source programs) for the incident. However, we believe the crucial reason is that the S-expression language (S-language), although used as intermediate language, was high-level enough for programming and even facilitating program construction. Indeed, the S-language is as high-level as the F-language (of F-expressions), since what gets changed through the "trivial" [McCarthy 1959, p. 2] translation is only notation, *not* abstraction as in later-developed systems that translate source programs in some high-level language to some low-level intermediate language like the JVM byte code [Lindholm and Yellin 1999] or the LLVM intermediate representation [Lattner and Adve 2004].

[12] McCarthy's expectation was fulfilled in the short-lived LISP 2 [Abrahams et al. 1966].

[13] The assembly language was SAP (Symbolic Assembly Programs) for IBM 704.

[14] The universal function, `eval` according to McCarthy [1978] but `apply` according to Stoyan [2008], was an F-expression.

[15] For other possible versions of the story, see [Stoyan 2008]. Whatever version, the story shows how difficult but also how important it is for theoreticians and practitioners to communicate.

3 Conclusion

This investigation into the early development of Lisp shows how the look and feel of the language was shaped by mathematics and mechanics. The language was designed for writing programs algebraically. The algebraic feel was reflected in functional expressions (or F-expressions). The intention to represent them, internally in the machine led to the introduction of the list data structure, and externally in an intermediate language to the invention of symbolic expressions (or S-expressions). Gradually, list got abstracted as a data type and S-expressions admitted into the source-level language.[16] After the first interpreter-based implementation of Lisp was running, S-expressions wan out as the preferred language for programming. In addition to technical and historical reasons, the incident could also be credited to the identical abstractive power of S-expressions with F-expressions.

The design presented in [1959] suggests that the very basic compound data type McCarthy wanted to include into Lisp was list, *not* pair. This suggestion was justified by McCarthy's adherence to the mathematical notion of sequence in AIM-1 through AIM-8. After all, to process lists was one of the design goals of the language. Moreover, two-element lists cover all possible use cases of pairs. Some people may try to defend the status of pairs by appealing to space efficiency (since when storing two elements, a pair uses one less `cons`-cell than a list) or to obscure data structures (such as circular lists that represent infinitely-repeated sequences). However, now that space is no longer a big issue and the functionality of circular lists can be simulated by non-circular ones with a loop (more precisely, with a jump back to the start at the end), the existence of pairs in the language has become obsolete.

Examining the origin of the binary variant, we sense a mathematician's commitment to symmetry and reductionism, which is yet another "influence of the designer on the design" but which Stoyan [1991] has probably overlooked. The rationale McCarthy explicitly gave for proposing the binary variant was that "the unsymmetrical status of `first` and `rest` may be a source of uneasiness" [1959, p. 17]. The one he implicitly held was naturally for simplifying the system by reducing lists to nested pairs. However, both rationales were weakened by the resultant system. The symmetry was never used. Instead, the asymmetry he tried to eliminate was reintroduced, not by constraint but by convention. The reduction led to the dilemma we have seen, where the programmer either does *nothing sane* to bear it or *something insane* to circumvent it. We believe a well-designed language should never put the programmer in such an awkward situation. One may propose including another set of manipulation functions, say `first`, `rest` and `combine` as defined in the earlier part of [McCarthy 1959], particularly for lists. However, it does not really solve the problem, only further complicates the system. If improper lists and circular lists are rarely used and can always be simulated in the rare case, it is better to kick them out to favor the common case. We urge designers of new Lisp dialects to discard pairs and

[16] The rationale for this admission will be covered in the second essay of this series.

return to lists as presented in [McCarthy 1959].

This concludes the essay. In the following series, we will reveal other mysterious aspects of Lisp such as the relationship between code and data, the existence of a meta-circular interpreter, etc. Lisp, as the first language that embraces and integrates ideas from three major theoretical bases of computation, namely Turing machine, lambda calculus and recursion theory, is a goldmine worth deep digging.

4 Acknowledgments

We appreciate Herbert Stoyan's donation of his collection on Lisp and AI to the Computer History Museum [CHM 2010]. We especially thank Paul McJones of the Computer History Museum's Software Preservation Group team, as well as many others, for collecting, preserving and presenting historical materials about Lisp [McJones 2005].

References

The herbert stoyan collection on lisp programming. Computer History Museum, Lot Number X5687.2010. Finding aid: `http://www.computerhistory.org/collections/catalog/102703236`, 2010.

Paul W. Abrahams, Jeffrey A. Barnett, Erwin Book, Donna Firth, Stanley L. Kameny, Clark Weissman, Lowell Hawkinson, Michael I. Levin, and Robert A. Saunders. The LISP 2 programming language and system. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA*, pages 661–676, 1966.

Paul Graham. The roots of lisp, 2001. URL `http://www.paulgraham.com/rootsoflisp.html`.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

John McCarthy. An algebraic language for the manipulation of symbolic expressions. Technical Report AI Memo 1, Artificial Intelligence Laboratory, Cambridge, MA, USA, October 1958a.

John McCarthy. Symbol manipulating language — revisions of the language. Technical Report AI Memo 4, Artificial Intelligence Laboratory, Cambridge, MA, USA, October 1958b.

John McCarthy. Some proposals for the volume 2 (v2) language. Letter to A. J. Perlis and W. Turanski, June 1958c.

John McCarthy. Symbol manipulating language — revisions of the language. Technical Report AI Memo 3, Artificial Intelligence Laboratory, Cambridge, MA, USA, September 1958d.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Technical Report AI Memo 8, Artificial Intelligence Laboratory, Cambridge, MA, USA, March 1959.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238, New York, NY, USA, 1961. ACM.

John McCarthy. History of lisp. *SIGPLAN Not.*, 13(8):217–223, 1978.

John McCarthy. Lisp - notes on its past and future. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages .5–viii, New York, NY, USA, 1980. ACM.

Paul McJones. History of lisp. Software Preservation Group, Computer History Museum, `http://www.softwarepreservation.org/projects/LISP/`, 2005.

Robert Muller. M-LISP: its natural semantics and equational logic (extended abstract). In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, pages 234–242, 1991.

Robert Muller. M-lisp: A representation-independent dialect of lisp with reduction semantics. *ACM Trans. Program. Lang. Syst.*, 14(4):589–615, 1992.

Herbert Stoyan. Lisp history. *Lisp Bull.*, (3):42–53, 1979.

Herbert Stoyan. Early LISP history (1956-1959). In *LISP and Functional Programming*, pages 299–310, 1984.

Herbert Stoyan. The influence of the designer on the design — j. mccarthy and lisp. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

Herbert Stoyan. Lisp: Themes and history. In *International Lisp Conference, ILC 2007, Cambridge, April 1-4, 2007*, page 8, 2007.

Herbert Stoyan. Lisp 50 years ago. In *Celebrating the 50th Anniversary of Lisp*, LISP50, pages 3:1–3:2, New York, NY, USA, 2008. ACM.